CH.6: Synchronization Tools

* Background

- Processes can execute concurrently

⇐ بما انه العمليات في منها بتم تنفيذها بنفس الوقت ممكن يأدي الاشي هاد لـ data inconsistency
إذا كان في shared data (يعني الداتا ما تكون متناسقة مع إنها من نفس المصدر)

- Concurrent access to shared data may result in data inconsistency.

⇐ مشان نحل هاي المشكلة بلزمنا آليات متبعة نحلها ونتبعها.
طبعاً مثال على المشاكل هاي هي مشكلة الـ consumer-producer.

⇐ عنا الـ shared variable هو الـ Counter ⇐ بدل على عدد الـ Buffers الي فيهم بيانات

⇐ لما الـ producer يعبي الـ Buffer بنتقل للـ Buffer الي بعده، بزيد الكاونتر واحد
طبعاً بشرط ما يتعدى حجم الـ Buffer

⇐ الـ Consumer هو بقرأ الداتا، لما يكون الكاونتر صفر معناها ما في داتا يقرأها، اما اذا لا، فالـ consumer بحفظ القيم الي بالـ Buffer وبنقص الكاونتر واحد لأنه خلص البيانات الي في هاد المكان انحفظوا

* Race Condition

هسا بالحالة الي فوق وين بتصير المشكلة؟ لما ييجوا اثنين يقرؤوا قيمة الكاونتر ويعدلوا عليها بنفس الوقت

- Consumer:
```
reg1 = counter
reg1 = reg1 + 1
counter = reg
```
[counter++]

- Producer:
```
reg2 = counter
reg2 = reg2-1
counter = reg2
```
[counter--]

reg1 = 5
reg1 = 5+1 = 6

(قرؤوها بنفس الوقت صارت مشكلة)

reg 2 = 5
reg 2 = 5-1 = 4

~~c = 5~~
~~c = 6~~
c = 4 → هي المفروض هون 5

⇐ المفروض يشتغلوا بشكل منظم مش اثنين يعدلوا ويقاطعوا بعض

يعني بامنفصل الـ Race Cond. هو الحالة الي بتوصل فيه العمليات للداتا المشتركة وبتقوم بالتعديل عليها بصورة متزامنة (يعني العمليات تصير تتسابق عشان تجيب الداتا)

⇐ الحل هو إنه كل عملية تنفذ بدون ما العملية الثانية تقاطعها

* Critical Section Problem

، لو عنا سيستم فيه عدد من البروسسز وكان كل بروسس عنده critical section،

الـ critical section هي الجزئية المشتركة بين عدة بروسسز

.if Each process has critical section segment of code:

* process may be changing common variables, updating table, etc.

⇒ when one process in critical section, no other may be in its critical section.

⇐ المفروض ما يكون في بروسس بالكريتيكال سكشن، ولا بروسس تدخل هاي المنطقة إلا بإذن من الـ entry sec.، إذا كان في بروسس ممنوع بروسس ثانية تدخل.

⇒ general structure

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

* Algorithm for Process P_i

```
do {
    while (turn == j);
        critical section
    turn = j;
        remainder section
} while (true);
```

- while (turn == j); ⇐ إذا المفتاح مع j بنقعد نستنى j يفلّت المفتاح (المفتاح الي بدخّلنا ويعطينا إياه)
- critical section ⇐ إذا لا بدخل على الكريتيكال سكشن [illegible]
- turn = j; ⇐ بعدين بنعطي المفتاح لـ j بس نخلّص
- remainder section ⇐ باقي الكود الي على P_i نفذه

. The critical section problem: The problem of ensuring that when one process is executing in its critical section, no other processes is allowed to execute in its critical section.

* Solution to critical-section problem (بدنا نفهم حلول نطبقها عليها الـ ٣ شروط)

. ① Mutual Exclusion

⇐ إذا البروسس Pi جوّا الـ critical sec. ممنوع ولا أي عملية تدخل هاي المنطقة

② Progress

⇐ إذا الـ sec. critical فاضية وفي بروسس بدها تدخل، لازم نتأكد إنه ماني حدا طلب الكريتيكال سكشن قبلها، إذا في بروسس اي طلبت أوّل إلها الأولوية وشي لازم إجي منها بتأجل (أي إلها الأولوية).

③ Bounded Waiting

⇐ لازم يكون في عدد محدود من مرّات الدخول للكريتيكال سكشن اي واحد (طالما طبعاً ماني حدا بالـ الكريتيكال سكشن) وإذا تجاوزت البروسس هذا العدد وقتها بضغطها لتجي بروسس ثانية تدخل وتطلع (يعني مارح نخلي بروسس تفل تستنى كتير بسبب إنه في بروسس ثانية قاعدة بتدخل وبتطلع)

* Critical-section Handling in OS

الـ approaches رح نعتمد على نوع الـ kernal

① Preemptive: allows preemption of process when running in kernal mode.

⇐ بتسمح إنه يصير للبروسس interrupt وهو قاعد بتشتغل

② Non-preemptive: runs until exits kernal mode, blocks, or voluntarily yields CPU

⇐ إذا البروسس اشتغل رح يضل يشتغل لحد ما يخلص كل شي بإيده (طبعاً هاي الحالة مافيها race condition في الكريتيكال صود.

* Peterson's Solution

↳ Two process Solution

⇐ هاد الحل بنفع في حالة يكون عنا بس عمليتين، هاد الحل بتفهم إنه يكون عنا:

⇒ Two shared variables: int turn; Boolean flag[2];

↳ Indicates whose turn is to enter the critical section

Array to indicate if a process is ready to enter the critical sec.

⇒ flag[i] = true (معناها البروسس Pi جاهزة عشان تدخل الكريتيكال سكشن)

*Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        reminder section
} while (true);
```

entry section: flag[i] = true; turn = j; while (flag[j] && turn == j);

exit section → flag[i] = false;

- flag[i] = true; → البروسس i بده يدخل الكريتيكال سكشن (البروسس جاهز يدخل الكريتيكال سكشن)
- turn = j; → هاد بكون Pi [illegible] الكريتيكال سكشن
- while (flag[j] && turn == j); → البروسس j شغالة ودورها (لهذا البروسس i تستنى تخلص)
- flag[i] = false; → (بس يخلص بحط ال flag تاعته false)

⟸ هنا بدنا نتأكد إذا هاد الحل حقق جميع الشروط الثلاثة؟

① Mutual exclusion is preserved

P_i enters CS only if:

flag[j] = false or turn = i

ال turn رح تكون إما لـ i أو لـ j مش رح تكون لالهم سوا بنفس الوقت ⟸ يعني مستحيل العمليتين يدخلوا على الكريتيكال سكشن بنفس الوقت.

② Progress requirement is satisfied

البروسس مستحيل توقف دور مش إلها في هاد الحل لأنها بتظل واقفة عند ال while loop عشان لحد ما البروسس الثانية ما تكون بدها الكريتيكال سكشن أو لحد ما يصير دورها.

③ Bounded-waiting requirement is met

البروسس الي بدها تدخل بتكون واقفة عند الوايل لوب وأول ما تطلع البروسس الثانية من بتدخل، يعني عدد المرات المسموح لكل بروسس تستخدم ورا بعض هو مرة واحدة على الأكثر.

* الحل هاد حقق ال ٣ شروط ✓

* Synchronization Hardware

⇐ في كثير أنظمة بتوفر دعم من الهاردوير لحل هاي المشكلة.

. All solutions based on idea of locking

⇐ الطرق هاي مبدأها قائم على فكرة قفل الكريتيكال سكشن

. Uniprocessors - could disable interrupts

⇐ بالأنظمة الي فيها بروسيسور واحد بنقدر نوقف الإنتربتس على العمليات
بس بالعادة هاي الطريقة مش فعالة إذا كان عنا أكثر من بروسيسور

. Modern machines provide special atomic hardware instructions.

⇐ فيه إنستركشنز معينة في البروسيسور بتمنع إنه يتم مقاطعة هاي البروسيس

* Solution to critical-section Problem Using Locks

```
do{
entry section ⇒ acquire lock        . بلزم القفل عشان البروسيس تدخل
        critical section
exit section ⇒ release lock         . بس يخلص بترك القفل عشان هاي يانا ياخذه
        reminder section
} while (true);
```

* test_and_set Instruction

```
boolean test-and-set(boolean * target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

⇐ هاد الفنكشن وظيفته إنه ياخذ القيمة الي دخلت ويغيرها بعدين يعطيها True
وبرجّع القيمة الي دخلت أصلاً.

1. Executed atomically
2. Return the original value
3. Set the new value of passed parameter to "TRUE."

* Solution using test_and_set()

```
do {
    while (test_and_set (&lock));
        critical section
    lock = false;
        reminder section
} while (true);
```

* Compare_and_swap Instruction

```
int compare_and_swap (int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

① Executed atomically
② Return the original value of "value"
③ set the variable "value" of the passed parameter "new_value" but only if "value" == "expected"

* Solution using compare_and_swap

```
do{
   while (compare_and_swap(&lock, 0, 1) != 0);   // initially = 0
        critical section
   lock = 0;
        reminder section
}while (true);
```

⟸ بتكون بالأول قيمة ال lock تساوي 0 ، فالفنكشن بغيرها لواحد وبترجع 0 فالشرط بصير فولس و البروسس بتدخل الكريتيكال سكشن وبس تخلص بتفلت ال lock وبتغيرها لصفر عشان حدا ثاني يدخل على الكريتيكل سكشن بعد ما هي خلصت وهكذا

* Bounded-waiting Mutual exclusion with test_and_set

⟸ ممكن البروسس تدخل جوا الكريتيكال سكشن وتطول هناك فممكن ما تحقق شرط ال Bounded-waiting.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
       key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i+1) % n;
    while ((j != i) && !waiting[j])
         j = (j+1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* reminder section */
}while (true);
```

- `waiting[i] = true;` ← (معناها إنه البروسس بده يدخل الكريتيكال سكشن)
- `while (waiting[i] && key)` ← initially = false
- `key = test_and_set(&lock);` ← key = false / lock = true
- `waiting[i] = false;` ← (دخل الكريتيكال سكشن خلص ما بستنى)
- `n` ← No. of processes
- `j = (i+1) % n;` ← (هون زي كأنه بروح يشوف البروسس اللي بعده)
- `j = (i+1) % n;` … `j = (j+1) % n;` ← هل البروسس اللي جنبي بده يدخل؟ إذا لا بروح أشوف اللي جنبه لحد ما يلاقي بروسس بده يدخل
- `if (j == i)` / `lock = false;` ← إذا ما حدا بده يدخل فـ رجّع ال lock فولس عشان إذا بروسس جت تدخل بعدين تلاقي ال lock فولس
- `else waiting[j] = false;` ← بنعطي j الصلاحية يدخل للكريتيكال سكشن

* Mutex Locks

بشكل عام، الحلول السابقة هي حلول معقدة، المبرمجين بالعادة ما بقدروا يوصلوا

- OS designers build software tools to solve critical section problem ⇒ Mutex Locks
- Protect a critical section by first acquire() a lock then release() the lock.
  - Boolean variable indicating if lock is available or not.

    ← إذا كان هذا المتغير 1، إذن الـ lock مش متاح في حدا بستخدمه

    ← إذا كان 0، إذن الـ lock مفتوح ومتاح للاستخدام
- Calls to acquire() and release() must be atomic.

  ⇒ usually implemented via hardware atomic instructions.
- But this solution requires busy waiting.

  ← إنها البروسس تظل تستنى ليصير الكريتيكال سكشن متاح (بعمل delay).

  ⇒ This lock therefore called a spinlock.

* acquire() and release()

بستنى ليصير متاح عشان يوخذه ويحوله لغير متاح عشان ما حدا ياخذه وهو صار

```
acquire() {
    while (!available);
        /* wait */
    available = false;
}
```

```
release() {
    available = true;
}
```

بيخلي الـ lock ياني متاح

```
do {
    acquire lock
        critical
    release lock
        reminder
} while (true);
```

* Semaphore

- Synch. tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

⇐ أداة مُزامنة متطورة أكثر، أفضل من الفكرة السابقة اللي هي الـ Mutex بحيث إنه البروسس بتخصوصها عشان يحققوا الـ synchronization.

- Semaphore S : integer variable

⇐ بس فنكشنين بقدروا يغيروا و يعدلوا على قيمة الـ semaphore

wait() and signal() ⇒ P() and V()

```
⇒ wait(S){
      while (S<=0);
        /* wait */
      S--;
  }

  signal(S){
      S++;
  }
```

* بتنقص من الـ S واحد طالما ما وصلت قيمة الـ S للصفر

* بتزيد قيمة الـ S بمقدار واحد

* Semaphore Usage

- Counting semaphore: Integer can range over unrestricted domain.
- Binary semaphore: Integer can range between 0 and 1.
  - ↳ same as mutex lock

⇐ السيمافور بقدر يحل مشاكل السنكرونايزيشن المختلفة.

- consider P_1 and P_2 that require S_1 to happen before S_2

```
synch = 0 (semaphore)
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

wait(synch); ⇒ مستحيل أنفذ S_2 إذا ما تنفذت S_1 (لأنه S_1 بتخلي synch = 1)

⇐ استخدمنا الـ semaphore هون عشان نرتب تنفيذ العمليات بالطريقة اللي بدنا إياها

* Semaphore Implementation

. Must gurantee that no two processes can execute the wait() and signal() on the same semaphore at the same time.

⇐ السيمافور الوحدة ممنوع أكثر من تو بروسس عم يكولوها بنفس الوقت.
يعني بنحط ال wait() , ال signal() بال critical section جواتها يعني.

⇐ باي الحالة رح يصير يكون فيه busy waiting جوا الكريتيكال سكشن بس كتابة الكود بتكون أسهل، وعملية الانتظار نفسها قصيرة.

* Semaphore Implementation with no Busy waiting

. with each semaphore there is an associated waiting queue.

. Each entry in waiting queue has two data items:

① Value (of type integer)

② pointer to next record

. Two operations:

① Block ⇐ رح نوقف البروسس P_i مثلاً وأحطها بال waiting queue

② Wakeup ⇐ نشيلها من الويتنغ كيو ونحطها بالريدي كيو

```
. typedef struct{
    int value;
    struct process *list;
  } semaphore;

. wait (semaphore *s) {
    s->value --;
    if (s->value < 0){
      add this process to s->list;   ( نحطها في الويتنغ كيو )
      block();
  }
```

```
. signal (semaphore *S) {
    S -> value ++;
    if (S -> value <= 0) {        ← قاعدة بتستنى
        remove a process P from S -> list;
        wakeup();                 ← بنطلعها من الويتنغ ليست
    }
}
```

* Deadlock and Starvation

Deadlock : two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting process

← يعني أكثر من بروسس قاعدين بستنوا بعض يصير عند بروسس ثانية قاعدة بتستنى أصلاً

. let S & Q semaphores = 1

P_0	P_1	S = ~~1~~	Q = ~~1~~
wait(S);	wait(Q);	0	0
wait(Q);	wait(S);	stuck here	
...	...		
signal(S);	signal(Q);		
signal(Q);	signal(S);		

. starvation – indefinite blocking

A process may never be removed from the semaphore queue in which it is suspended.

← بضل تأجل البروسس لأنه في بروسس ثانية أهم منه.

. Priority Inversion

scheduling problem when lower-priority process holds a lock needed by higher-priority process

. solved via priority-inheritance protocol

← بروسس أقل أهمية معه الـ lock الي بتاجه بروسس أعلى أهمية منه.

* Problems with semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) ... wait (mutex)

    ⇐ بالعادة القيمة بتكون 1 وهي بولين، ما بنفع نخليها 2!
  - wait (mutex) ... wait (mutex)

    ⇐ هاي الحالة بتسبب deadlock.
  - Omitting of wait (mutex) or signal (mutex) (or both)

    ⇐ لو نسيت أي منهم أو الثنين هاد غلط جداً"
- Deadlock & starvation are possible.

* Monitors (⇐ عنا في الديدلوك، زي مشاكل بيسبب أوقات السيمافورز لأنه)

A high-level abstraction that provides a convenient and effective mechanism for process synchronization.

⇐ في حال إنه البروجرامر نفسه فهم البروسس بطريقة خطأ، ممكن يصير في مشاكل، لهيك الـ monitor موجود (مش شرط فيه إنه البروسس تكون بيرفكت عشان ما يصير مشاكل زي في حالة الـ semaphores)

- The monitor itself programmed to provide the mutual exclusion, and when the processes wants to access some shared data, they will do it via the monitors and hence the monitor will provide the mutual exclusion in order to achieve the process synchronization.

⇐ لما البروسس بدها أي شي بتوصله عن طريق الـ monitor.

- Monitor contains the declaration of variables & the bodies of procedures that operates on those variables.
- monitor monitor_name {

```
monitor monitor_name {
  // shared variables declarations
  procedure P1 (...) {...}
  procedure P2 (...) {...}
  procedure Pn (...) {...}
  Initialization code (...) {...}
}
```

⇐ البروسيجرز هدول هم عمليات بتم تنفيذها على الـ shared variables

(تعطي القيم الأولية للفاريبلز إلي بنستخدمها) ← Initialization code

⇐ المونيتورز بوفروا حماية أعلى لأنه بس البروسيسز الي جوا الـ Monitor بقدروا يوصلوا للفاريبلز وما حدا غيرهم بقدر يوصل لهدول الـ shared variables.

- Only one process at a time can be active within the monitor

* Condition Variables

⇐ عشان نحقق الهدف من وجود الـ monitor لازم نضيف شغلة مهمة عليه وهي الـ condition construct.

condition x, y;

⇐ في بس عمليتين مسموح يصيروا على الـ condition var. وهم:

X.wait() : means that the process invoking this operation is suspended until another process invokes X.signal()

X.signal() : resumes one of processes (if any) that invoked x.wait().

لو في بروسس بدها تستخدم X وهي مش متاحة بتستخدم X.wait()، وبتضل هاي البروسس تستنى فيها لحد ما البروسس الي قاعدة بتستخدم فيها تنفذ X.signal() (والي معناها إنه X جاهزة).

* Condition Variables choices

⇐ البروسسز ما بصير يتنفذوا بنفس الوقت، لهيك إذا في بروسس بتستنى بـ X مثلاً وبروسس ثانية استخدمت X.signal() بس لسا ~~[illegible]~~ بدها تكمل شغلها، في عدة خيارات لهالإشي:

P ⇒ X.signal() Q ⇒ X.wait()

- signal and wait : P waits until Q either leaves the monitor or it waits for another condition.
- signal and continue : Q waits until P either leaves the monitor or it waits for another condition.

* Monitor Implementation Using Semaphore

```
semaphore mutex;
semaphore next;
int next_count = 0;

wait(mutex);
   ...
     body of F;
   ...
if (next_counter > 0)
    signal(next);
else
    signal(mutex);
```

البروسس بستنى لحد ما القفل أو المفتاح يصير معه
بصير ينفذ الي بده ينفذه ، إذا البروسس
بدها تستخدم الأشياء كمان مرة بتستخدم
signal(next) ، إذا لا بتنهي شغلها
مع ال mutex عن طريق ال
signal(mutex).

* Monitor Implementation – Condition Variables

```
semaphore x_sem;   بداية جزء
int x_count = 0;
```

لكل كونديشن فاريبل في عنا

```
x.wait():
    x_count++;
    if (next_counter > 0)
        signal(next);
    else
        signal(mutex);
    wait(x_sem);
    x_count--;
x.signal():
    if (x_count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
    }
```

⟸ عيب هاي فاهم اشي ؟

* Resuming Processes within a Monitor

⇐ السؤال وإذا في أكثر من بروسس قاعدة بتستنى وتم تنفيذ X.signal() ، أي وحدة نعطيها ، نخليها تشتغل؟

- FCFS frequently not adequate.

⇐ في عنا اشي اسمه conditional-wait بكون بهاد الشكل:

```
X.wait(c);
```

بحيث إنه c هو أولوية هاي البروسس

- Process with lowest number (highest priority) is scheduled next.

* Single Resource allocation

⇐ الحصول على الريسورس اي بدنا إياه بكون بناءً على الأولوية اي هي أصلاً بتحدد قديش أكثر وقت العملية فيه هاد الريسورس. بدها تستخدم

```
R.acquire(t);
    ...
access the resource;
    ...
R.release();
```

⇒ R is an istance of type Resource Allocator.

```
monitor ResourceAllocator {
  boolean busy;
  condition x;
  Void acquire (int time) {
        if (busy)
           x.wait (time);
        busy = TRUE;
  }
  Void release(){
        busy = False;
        x.signal();
  }
  initialization code () {
      busy = False;
  }
}
```

⇐ إذا الريسورس مش متاح بتضل تستنى فيه وبس تمسكه بتخلي ال Busy ترو ع طول

⇐ بس العملية تخلص بتطلعه وراح المتغير.